A pictorial approach to understanding MPI program behaviour

Victor Eijkhout*

December 8, 2022

1 Motivation

One of the biggest difficulties facing a novice MPI programmer is to devise a 'node program' that, when executed on each of a set of processors, achieves a certain desired overall behaviour. Conversely, when the programmer has written an SPMD program, and its execution is not as desired, there is the difficulty of understanding how the individual node executions add up to the global behaviour.

The first place where this local/global dichotomy is likely to show up is in a deadlock resulting from the use of blocking calls. More deceptively, a program can also execute and terminate correctly, but run far slower than intended.

Consider a simple example, where we have an array of processors $\{P_i\}_{i=0..p-1}$, each containing one element of the arrays x and y, and P_i computes

$$\begin{cases} y_i \leftarrow y_i + x_{i-1} & i > 0 \\ y_i \text{ unchanged} & i = 0 \end{cases} \tag{1}$$

Since this operation is not a recurrence, we expect a parallel execution to run in essentially $O(1)$ time.

A naive MPI implementation could be written along the following lines:

- If I am processor 0 do nothing, otherwise receive a y element from the left, add it to my x element.
- If I am the last processor do nothing, otherwise send my y element to the right.

If we use blocking communication, a send instruction does not finish until the sent item is actually received, and a receive instruction waits for the corresponding send. In our example, the resulting execution will then have all processors waiting to receive, except for processor 0 which

* eijkhout@tacc.utexas.edu, Texas Advanced Computing Center, The University of Texas at Austin

sends to 1. Processor 1 can then complete its receive, and send to 2, et cetera. We see that what should have been a perfectly parallel operation actually has a running time of $O(P)$.

It is this sort of phenomenon that we aim to understand and prevent.

2 Pictorial analysis

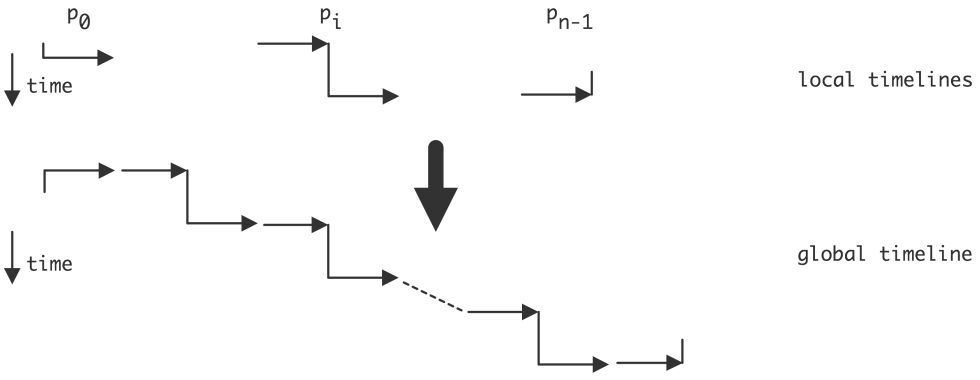
The problem as signaled in the introduction is that, informally, we have trouble 'visualizing' the whole program execution in terms of the node executions. In this note we propose a way of actually *visualizing* the execution.

We start by picturing the node programs:

- Time is pictured vertically, so each node program is based on a vertical line.
- Sends and receives are depicted by arrows attached to the timeline: a receive is indicated by an arrow pointing towards the timeline, and a send by an arraw pointing away.
- If we want to emphasize a local operation taking place, we can put a marker on the vertical timeline, as will be done in the next section.

The p_i program in the following picture depicts a node program that executes a receive call followed by a send call. The send call can use data that was computed from the received data, but that is not necessary. For instance, the motivating example of the introduction had no such propagation of information.

The overall execution can now be constructed by matching up sends and receives of node programs. In this simple example, it's easy to see that we get serialized execution:



A note on how to interpret this picture.

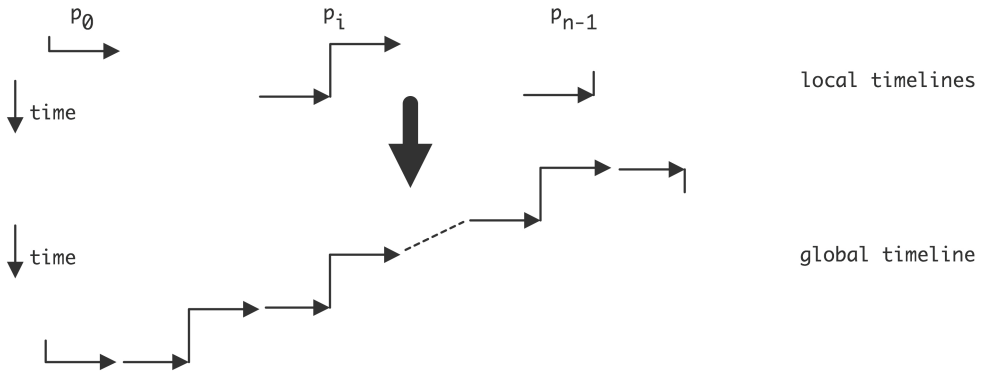
- The global timeline in this picture is obtained from fitting together the node programs with the local timelines:

- The global timing of the execution of a node program results from how its receive operations synchronize with the send of another processor execution.
- We again note that while this picture looks like there is global information flowing through the processors, this need not be the case.

What if we reverse the send and receive operations?

- If I am not the last processor, send my x element to the right;
- If I am not the first processor, receive an x element from the left and add it to my y element.

The following illustration shows that again we get a serialized execution, except that now the processor are active right to left: the final processors starts by accepting a receipt from the one before; the penultimate processors concludes its send and accepts a receive from its left neighbour, et cetera.



If the algorithm in equation **??** had been cyclic:

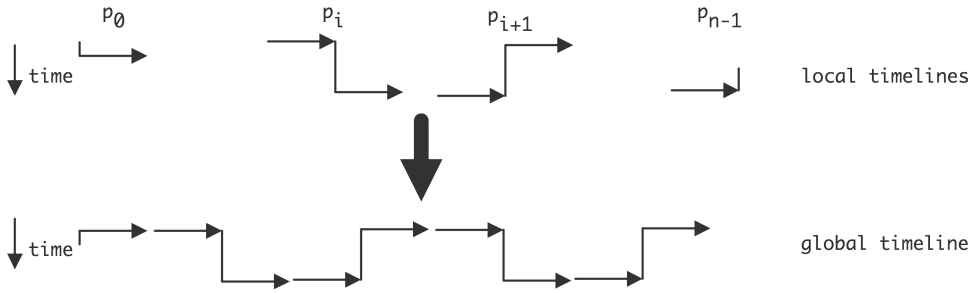
$$\begin{cases} y_i \leftarrow y_i + x_{i-1} & i = 1 \ldots n-1 \\ y_0 \leftarrow y_0 + x_{n-1} & i = 0 \end{cases} \tag{2}$$

the problem would be even worse. Now the last processor can not start its receive since it is blocked sending x_{n-1} to processor 0. This situation, where the program can not progress because every processor is waiting for another, is called deadlock. In our picture this shows as the inability to match the first to the last processors: there is no global execution consistent with the node program.

One solution to getting an efficient code is to make as much of the communication happen simultaneously as possible. After all, there are no serial dependencies in the algorithm. Thus we program the algorithm as follows:

- If I am an odd numbered processor, I send first, then recieve;
- If I am an even numbered processor, I receive first, then send.

If we illustrate this we see that the execution is now parallel.



We will see another solution in section **??** below.

3 A non-trivial example

Let's now apply this pictorial analysis to developing a non-trivial communication scheme.

The Gauss-Seidel scheme for a tri-diagonal matrix computes

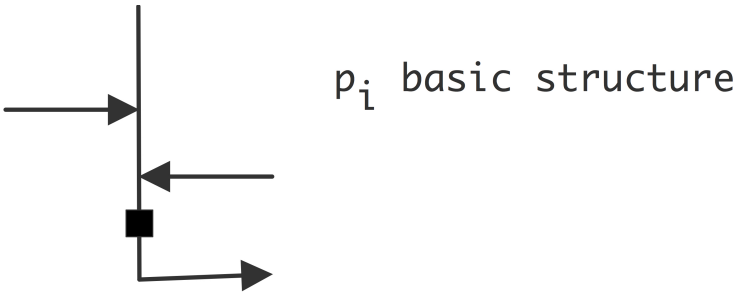
$$\forall_i \colon x_i^{(n+1)} = \alpha_i^{-1}(\beta_i - x_{i-1}^{(n+1)} - x_{i+1}^{(n)}) \tag{3}$$

Since the computation on component i involves the left and right neighbours $i-1$ and $i+1$, we know that the code has at least the following structure:

- Receive from the left,
- Receive from the right
- Compute.

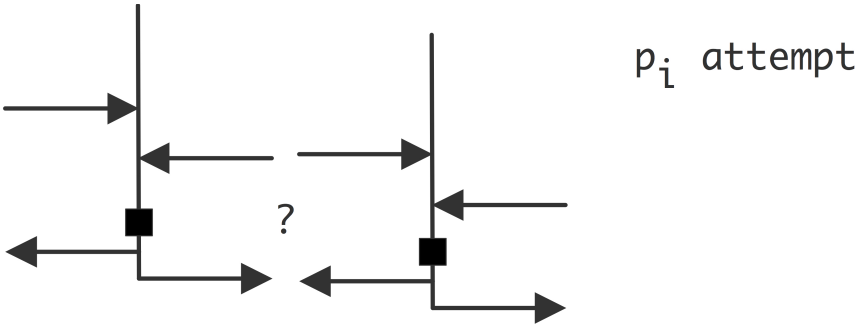
It is clear that the computed result also has to be sent to the right. Also, corresponding to the 'receive from the right' there has to be a 'send to the left'. The remaining question is how to sequence this so that no deadlock ensues.

First we make a picture of the basic structure of the node program, including the three communications that we agree on:

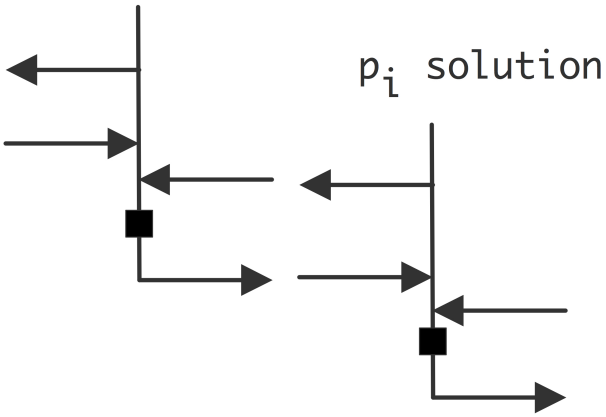


If we do the left send late we get the following picture, and we see that deadlock ensues because

the sends and receives between neighbours don't match up:

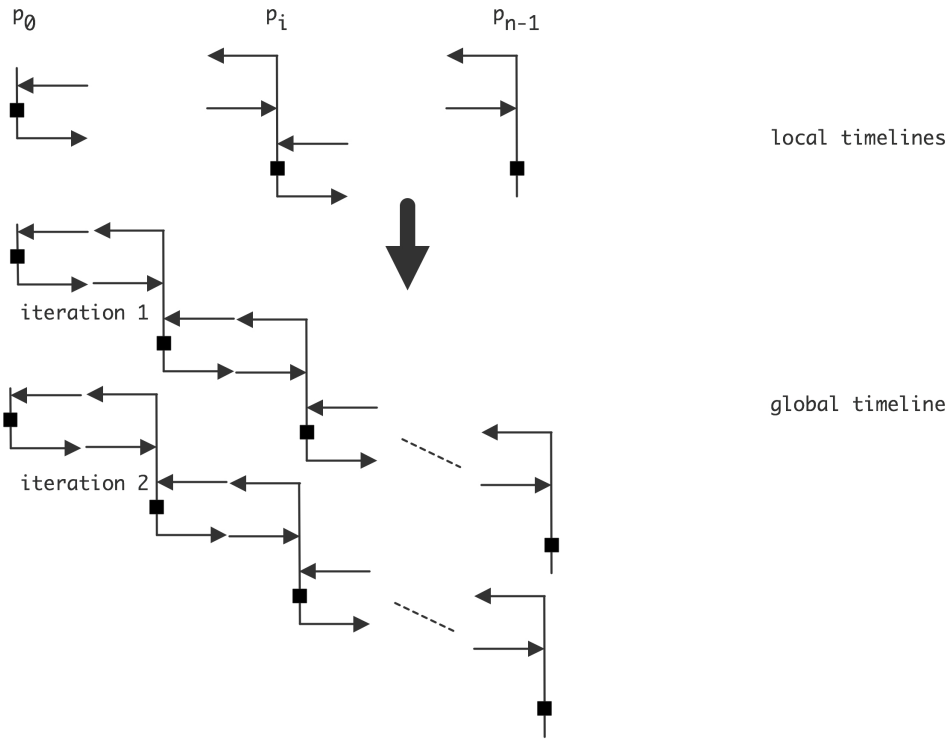


On the other hand, if we send left before receiving from the right, we see that that the pieces fit:



And since all processes repeat their bit of send/recv code we get a picture that shows the

pipelined behaviour:



4 An example with the MPI sendrecv function

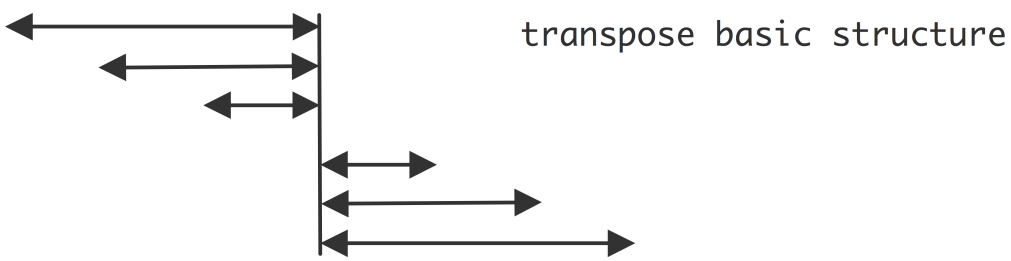
Knowledgeable readers will have wondered 'can these problems not be solved by using `MPI_Sendrecv`?' That routine can indeed solve deadlock and serialization problems in cases where the processes can be paired up, such as with the initial example that started this discussion. However, even with the Sendrecv call there are questions about parallel behaviour.

For instance, consider the following code for doing a data transposition, meaning that each processor has to exchange data with every other.

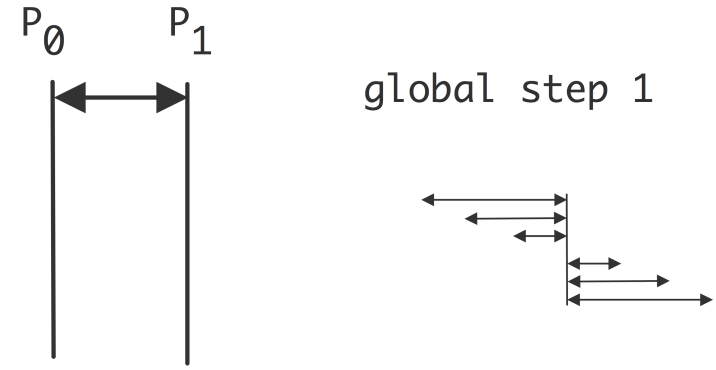
```
for (p=0; p<mytid; p++)
  MPI_Sendrecv( .... /* to: */ p, /* from: */ p ... )
for (p>mytid; p<ntids; p++)
  MPI_Sendrecv( .... /* to: */ p, /* from: */ p ... )
```

Does even the knowledgable reader immediately see what the sequential time of this algorithm is? Clearly the number of steps is at least the number of processors, but is it more? Do we get a serialization behaviour and is the running time of the order of the processors squared?

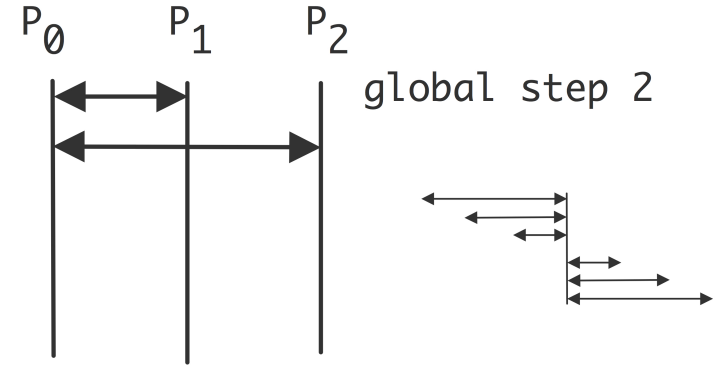
Let's start by picturing the basic structure:



If we try to match up processes, we see that everyone has an arrow pointed at 0, but zero only matches with 1:

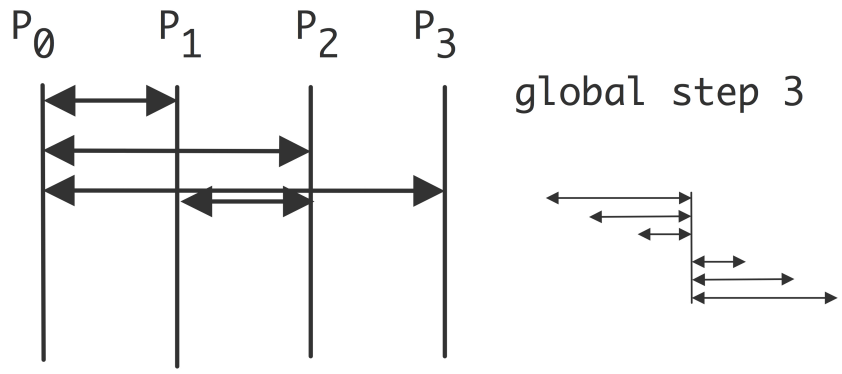


Next 0 communicates with 2:

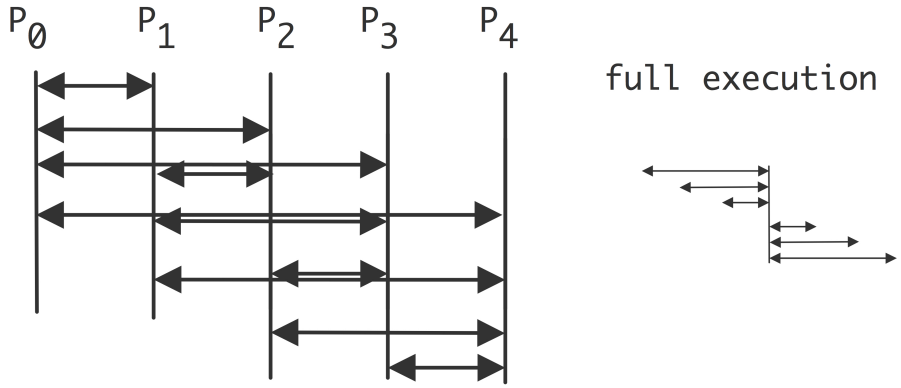


In the next step, process 0 continues to exchange data further to the right, in this case with 3. Additionally, process 1, having completed its exchanges to the left, can now start exchanging data to the right, here with 2. We indicate this simultaneous behaviour by putting the arrows

close to on top of each other:



We see that in every step a new processor becomes active. Finishing the picture for the full algorithm, we that processors are gradually included in the execution, and also gradually drop out again:



We conclude that the algorithm has a running time of $2(P-2)+1 = 2P-3$. (Did you guess this $2P$ complexity in the beginning of this section?)

5 Another Sendrecv example

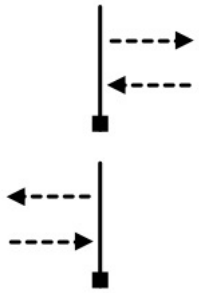
The MPI standard states that 'A send-receive operation is very useful for executing shift operation across a chain of processes.' Indeed, the problem signalled in section **??** is easily solved by using a send-receive operation.

The problem of a process gathering data from both neighbors is easily implemented as two shifts: one to increasing, and one to decreasing process numbers. But what if we implement this neighbor gather as

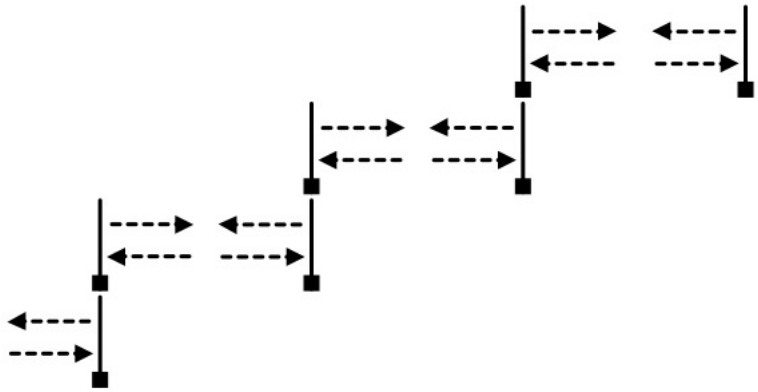
```
Sendrecv( from=p+1, to=p+1 )
Sendrecv( from=p-1, to=p-1 )
```

One might think that because send-receive is implemented as non-blocking calls this will again be perfectly parallel. However this is not so. The fact that there are two (implicit) waits induces a sequentialization.

We can depict the basic block as:



Putting the basic blocks together gives a global picture (note that the dotted arrows are non-blocking so their sequence needs not match up):

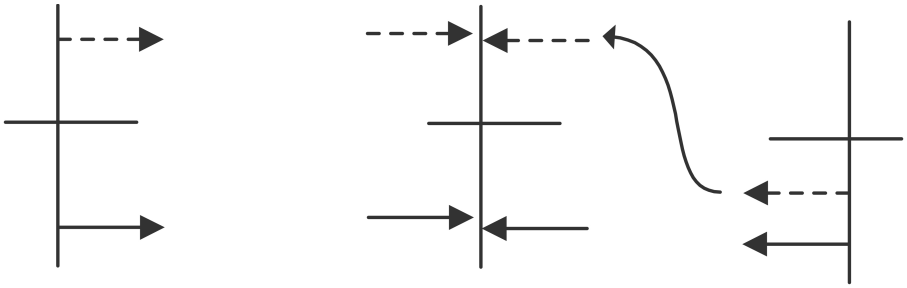


6 An example with non-blocking sends

We depict non-blocking operations by a dotted arrow and a solid arrow for the Wait operation. With that, consider an interaction between three processors. In this example, process 1 posts a wildcard receive before a barrier; of the two other processes one posts a send before the barrier, the other one after.

```
// proc 0                  // proc 1                          // proc 2
Isend( /* to: */ 1 )       Irecv( /* from: */ ANY_SOURCE )    Barrier()
Barrier()                  Barrier()                          Isend( /* to: */ 1 )
Wait                       Wait                               Wait
```

You would expect that on process 1 the receive before the barrier can only be paired with a send before the barrier, that is, the send on process 0. However, the standard allows for the transfer operation to happen only at the Wait call. That enables the following pairing:



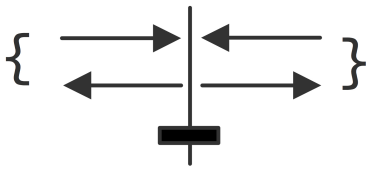
In other words, data seems to travel ‘backwards through the barrier’.

7 Repeated operations

Above, in section **??** we concluded by showing the resulting behaviour if a node program is repeatedly executed. In this section we will show how we can optimize communication behaviour if we know in that there are multiple ‘tasks’ to be executed.

Let us for now define a *task* as processor-local work that operates partly on remote data. This means that our node program consists of

- Some sends and some receives, in an order to be determined; followed by
- Local computation.



The curly brace denotes that the send and receive can be done in either order. We only draw a send and receive from left and right, but in general there can be any number of neighbours.

unfinished section

8 Discussion

Understanding the behaviour of an MPI program as it results from a given SPMD program code is not trivial. While experienced programmers may develop such an intuition, or alternative have a mental ‘cookbook’ of recipes as well as a list of ‘counter-examples in parallel programming’, for beginning programmers this can be quite a stumbling block.

This note demonstrated a pictorial technique for visualizing MPI program behaviour that can prove enlightening in some cases. Of course this technique falls far short of an actual theory of inferring global behaviour from the SPMD program code. Also, the limits of two-dimensional drawing paper probably prevent this technique from being applied to algorithms that use any but a linear arrangement of processes.

However, we hope that the examples given here show at least *some* utility.